

Autonomous Objects: A Natural Model for Complex Applications*

Alfons Kemper*

Peter C. Lockemann⁺

Guido Moerkotte⁺

Hans-Dirk Walter⁺

*Fakultät für Mathematik und Informatik
Universität Passau
Innstrasse 33
D-8390 Passau, F.R.G.
kemper@fmi.uni-passau.de

⁺Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
D-7500 Karlsruhe, F.R.G.
moer|walter@ira.uka.de

1 Introduction

In the classical—and today still prevailing applications—of the business and administration worlds databases are primarily thought of as *passive* repositories of more or less vast amounts of information that should be available at the fingertips of human clerks, specialists or decision makers. It is these humans that react to outside stimuli, observed situations, orders, or external events to swing into action and in turn take the initiative in accessing their databases. Even the more recent object-oriented techniques address issues that still are inherently passive. Their emphasis is, by providing operations and encapsulation, on easing application design, consistency maintenance, and implementation flexibility. By adding strong typing, type hierarchies and inheritance, they also achieve a more modular system construction, better adaption to changing environments, and a more economic programming style.

Over the past years it has been recognized that at least in the world of office automation or inventory control the premise of *passivity* does not hold any longer. Actions are not solely stimulated by outside events but should also be due to situations that can be formulated beforehand, e.g., those that exhibit a regular or prespecified temporal pattern or describe distinguished states of the miniworld. For example, in inventory control the stockroom clerk may be informed automatically whenever the quantity on hand of an article falls below a prescribed limit. If one manages to have an up-to-date and complete image of the miniworld as a database, one may leave the automatic notifications to the database. Databases with these capabilities are often referred to as *active* DBMS. The prevalent language constructs for active databases are the event-trigger concept and the event-condition-action rules. In these approaches the events and conditions are specified in declarative form and globally added to the database. This, actually,

does not conform with the object-oriented paradigm which is based on object *encapsulation*.

A different—and more natural—approach in an object-oriented environment is proposed here. We make the individual objects *autonomous* “beings” in the database. We provide an object with facilities to go, after recognizing events and exceptional states, into action all by itself and often without outside interference. Such a mechanism seems most natural in situations where a database reflects a technical miniworld of, say, numerically controlled machine tools, robots, autonomous transport vehicles, etc., that is a factory floor and the associated stockroom and transport facilities. For example, the visual system of a robot may recognize the geometry and position of an oncoming workpiece and send the corresponding message to the informational object controlling the robot. The informational control object will then retrieve the corresponding manufacturing description of the workpiece together with all the information for controlling the robot motions from some other (database) objects and send it to the robot. It may also inform the following station of the upcoming piece, prompt an automatic vehicle to move to the stockroom to fetch a part and to transport it to the following station for the next assembly step, and send information on the fact that the workpiece has reached the robot to a control panel for display. This example highlights that there are a multitude of tasks to be carried out in parallel and mostly independent of each other. Independent activities of systems traditionally come under the notion of *process*. In other words, to perform independent actions a database system must be triggered into establishing processes.

The objective of the GOM project is to study databases that support technical activities such as the ones described in the last paragraph. Our previous discussion suggests that such databases should combine three concepts, *object-orientation*, *activity*, and *process* into a single coherent concept which we shall henceforth refer to as *autonomous object*. Ac-

*This work was supported by the German Research Council (DFG) under contracts Ke 401/6-1 and SFB 346.

cordingly, GOM provides the following support in a homogeneous and coherent framework:

Objects We follow the usual conventions for the notion of object. Its essential features are identified as: *object identity*, *object sharing*, the association of a specific *structure* and, possibly, a set of operators (the *behavior*) for accessing and manipulating the structure, the *classification* of structurally and behaviorally similar objects in types (or classes), the *subtyping* of types in order to inherit or refine their structure and/or their behavior, and the *encapsulation* of objects. We have developed an object oriented database system, called (passive) GOM that provides the above detailed features. Particular emphasis in the design of (passive) GOM was placed on strong typing [15, 13] and optimization [11, 12, 14].

The prevailing approaches to object-orientation assume that an object remains entirely inactive until one of its associated operations is explicitly invoked by some other (client) object. It then becomes active and immediately executes the request. After completion it becomes inactive again. Thus, there exists only a single-thread-of-control with its associated master/slave relationship within a collection of passive objects. It all boils down to an invocation mechanism equivalent to the procedure call.

Activity We call *activity* the ability to perceive and react on special situations. These are:

- Messages that arrive from other sources.
- The passing of predetermined time points, e.g., elapsed time intervals or wake-up calls from clocks.
- The detection of distinguished states or state transitions.

Often these situations are subsumed under the general term of “event.” Note, however, that both the mode of perception and the actions to be taken may differ in the three cases.

Associating activity with objects necessitates a number of restrictions. Due to encapsulation, monitoring can involve only states internal to an object. Further it makes sense to associate the perception mechanism with the entire object and let the object then decide which operation to perform. The mechanism, then, appears to “guard” the entrance to the object, hence we shall refer to it as guards (the similarity to Dijkstra’s guards [6] is no accident).

An active object clearly does away with single-thread-of-control because the object may come to live independently of what goes on elsewhere. However, one could still impose a rather strict control regimen by limiting the object to notifying some central

agency of its preparedness to act, and to leave it to the agency to schedule its action.

Processes A process is a meaningful sequence of elementary actions. It consumes time and consumes or occupies resources, and it has an observable effect. Such an effect may include changes to some global state, and/or messages that are sent off, e.g., to provoke activities elsewhere. One can distinguish several process states. Usually four states are identified: created, active, suspended, and completed.

Basically, processes operate entirely independently of one another, i.e., they are subject to truly multiple-thread-of-control. In particular, processes may execute concurrently.

To combine processes and objects, two significant alternatives exist. First, there is just one master process for each object. Second, one process is associated with each operation of an object. The former seems more in tune with the object philosophy because it allows an object to retain full control over its operations. The latter, however, would allow an object to respond to several stimuli concurrently. Concurrency within one object could be realized by several lightweight processes associated with the one master process.

Interaction in an object world seems to preclude cooperation (via shared data items) because of encapsulation. This leaves communication as the sole mechanism for interaction of objects. In turn, this requires to set up communication channels between objects.

Autonomy Roughly speaking, autonomy of an object means that the object has full control over its destiny. If we follow our previous discussion, control entails two aspects: activity and processes. (For a further characterization of the notion of autonomy see [10].)

Control over processes subsumes control over the state at any given time. Consequently, an object must be able to decide on its own which state transition to apply and when. This could be seen as a further argument in favor of a single process per object. Quite clearly, autonomy requires a permanent existence of such a process, hence it should be created and started on creation of the object. If the master process suspends itself the object must include a mechanism for resuming it.

Control over activities entails guards with a certain decision making capability. Typical decisions are to select which stimuli are of interest, to accept or ignore stimuli, to process them in an order determined by the object itself, to delay a response, to batch the processing of several stimuli. We assume, however, that decision making is not haphazard but follows a

given strategy which we refer to as the computation model of the object.

To summarize, an autonomous object has all the qualities of a traditional object, constitutes a process and possesses a set of guards which perceive stimuli and prompt reactions according to some computational model. Consequently, the description of an autonomous object includes the descriptions typical for objects, and additionally a description of the guards which we refer to in the remainder as the *behavioral map* of the object. This concept was strongly influenced by [2]. Autonomous objects were first proposed in [17] and [20, 4] which itself are influenced by the design of the ACTORS language [1]. Another implemented approach to active objects is described in [5]. Recently, Liu and Mersman [16] devised an activity model for object-oriented databases.

Our current approach to object activity is to view an autonomous object as a strictly sequential process, that is, to have just one master process which allows the processing of only one action at a time. We do not—currently—support intra-object parallelism. Furthermore, we limit our discussion here to one fixed computational model associated with an autonomous object. This model is based on indeterministic selection of the guards in the behavioral map for evaluation. The indeterministic selection process of the guards may be “tuned” by associating relative priorities with the guards.

2 The Generic Object Model GOM

In this section we describe the basic concepts of our object model GOM—for now, without the extensions for autonomous objects. The basic GOM concepts are very similar to the essential features identified in the “Manifesto” [3].

We will illustrate our discussion on a drastically simplified logistics control application which, subsequently, serves as a basis to discuss the shortcomings of a passive object model in more detail. In subsequent sections we will introduce the extensions for autonomous objects and then revise the logistics control application in order to achieve a more natural modeling using autonomous objects.

2.1 Overview of GOM

GOM is an object-oriented data model that allows the database designer to model persistent application-specific objects which incorporate their *structural* and *behavioral* description. Like most other object-oriented models GOM allows to classify objects and define a template, called *type*, from which

individual objects can be created by instantiation. Additionally, there exists a subtype relation together with inheritance which, however, is not further discussed in this paper.

The structural description of a type can be either one of the following: A *tuple* consists of a collection of typed attributes. The tuple constructor is denoted as $[a_1 : t_1, \dots, a_n : t_n]$ for unique attribute names a_i and type names t_i . A *set* is denoted as $\{t\}$ where t is a type name. A set of this type may only contain elements of type t or subtypes thereof. A *list* is denoted as $\langle t \rangle$. Lists are analogous to sets except that an order is imposed upon the elements (and duplicate elements are possible).

A simple tuple structured object type *Bolt* is sketched below (the keyword **persistent** initiates the inclusion of the type into the database schema):

```
persistent type Bolt supertype WorkPiece is
  body [Length: float,
        Thickness: float]
  operations
    declare weight:  $\rightarrow$  float code weightCode;
    ...
  implementation
    define weightCode is
    ...
end type Bolt;
```

In this definition we assume that the supertype *WorkPiece* has been defined elsewhere. All of its attributes and operations are inherited by the type *Bolt*. Additional operations are incorporated into the object type *Bolt* by specifying the signature in the **operations** clause and providing the implementation (possibly under separate name) in the **implementation** clause. In GOM it is also possible to *refine* inherited operations.

An example of a collection-valued type is *BoltSet* which is defined below:

```
persistent type BoltSet is
  body { Bolt }
  operations
  ...
  implementation
  ...
end type BoltSet
```

GOM provides a large number of built-in operations on collection types. Among the operations on sets are: *cardinality*; *union*, to union two compatible set-valued objects; *retrieveElems*, to remove a specified number of elements from the argument set and return them as a new set of the corresponding type; *insert* and *delete*, to add a new element to respectively to delete an element from the set the operation is invoked on.

2.2 The Logistics Control Application

Having sketched the GOM object model we can now attempt a first implementation of a very simplistic logistics control application. We will merely outline an object type called *BoltBox* which responds to two operations:

- *consume* to retrieve a specified quantity of *Bolts* from the *BoltBox* object and
- *supply* to furnish a *BoltBox* object with new *Bolts*.

In this respect instances of the *BoltBox* object type and objects retrieving bolts from them are comparable to the consumer/producer example which is one of the archetypes of a process communication pattern [2].

The example application is sketched as follows (the keyword **self** refers to the object instance in which the particular operation is invoked):

```

persistent type BoltBox is
  body [MinQOH: int;    !! threshold for reordering
        Box: BoltSet;   !! container for Bolts
  operations
    declare consume: int → BoltSet code consumeCode;
    declare supply: BoltSet → void code supplyCode;
    declare runninglow: → BOOL code runninglowCode;
  implementation
    define consumeCode(quantity)
    begin
      if (quantity ≤ self.Box.cardinality)
        return self.Box.retrieveElms(quantity);
      else ...    !! error: not enough Bolts available
    end define consumeCode;
    define supplyCode(Bolts)
    self.Box.union(Bolts); !! insert new supply
    define runninglowCode
    return self.Box.cardinality ≤ MinQOH;
  end type BoltBox;

```

Now, the semantics of the above types can briefly be outlined. *BoltBox* encapsulates two attributes: *Box* which serves as a “container” for *Bolts* and *MinQOH* which is the threshold value at which new *Bolts* should be reordered. Bolts can be retrieved from an instance of *BoltBox* by invoking the operation *consume*. Producers can *supply* new *Bolts* by invoking the operations *supply* with the appropriate argument of type *BoltSet*. In order to enable other objects to check whether the bolts of a *BoltBox* instance are running low the respective operation *runninglow* is declared and defined. This test may be performed, e.g., from time to time or at each invocation of *consume*.

The above model of the *BoltBox*, which is a highly shared object in our (envisioned) logistics control system, bears several severe disadvantages which will be highlighted in the following discussion.

In real-life applications there would, of course, be more than one consumer and/or supplier for the same *BoltBox* instance. These objects may be distributed within the factory and typically work in parallel. This necessitates—under the conventional object paradigm as used in the above programmed example—a centralized scheduler, which is sometimes called the application *script*. The task of this script is to transform the multitude of parallel activities—e.g., *consume* by many consumers, *supply* by several producers, monitoring the object base state to detect abnormal situations—into the single-thread-of-control execution model. As in [10] we argue that the realization of this centralized, global application script becomes unmanageable in real-life CIM applications—as well as in other complex applications.

Furthermore, the realization of sophisticated, co-operative behavior of objects in response to requests from other objects is difficult. An example may illustrate this: a large *consume* request that cannot be fulfilled due to lack of *Bolts* should be postponed until a new supply arrives. However, in the meantime small *consume* requests should be granted. An autonomous *BoltBox* could have postponed the large *consume* request in order to wait for new incoming supply. There is no need that a consumer should have noticed this autonomous decision of the *BoltBox* object. Under single thread-of-control this has to be explicitly programmed into the application script.

One of the hardest problems in developing large software products is the integration of several complex subsystems into a single system. Under single-thread-of-control the integration of two— independently realized—subsystem means to develop out of two complex application scripts a single, global new one. The complexity of this task is one of the main reasons that—up to now—despite enormous efforts undertaken there still does not exist an “integrated” CIM solution. The same argument applies if a new complex subsystem has to be added to a set of existing subsystems. Thus extensibility in the large becomes a strong problem also present when using current object-oriented systems even though they advertise being easily extensible. But this extensibility can, at best, be termed extensibility-in-the-small; it does not provide the much needed support for configuring large object-oriented systems.

3 Autonomous GOM Objects and the Map

In order to account for autonomous behavior we—clearly—have to abandon the master/slave relationship incurred by procedure invocation. Therefore, we introduced “true” message passing in the sense that the receiver is autonomous, i.e., it has self-

determination about when and how it wants to react on the message. On the sender side we have to introduce the capability of sending a message, and on the receiver side we have to facilitate buffering and interpretation of incoming messages. To account for object autonomy a construct called **behavioral map** is introduced. The behavioral map allows to specify the object's autonomous behavior, i.e., the actions to be taken depending on:

1. the pending requests for executing operations from the outside world, e.g., other objects or database users
2. time events, e.g., certain operations that are invoked on a periodical basis
3. the current state of the object

Even though the behavioral pattern of an object is influenced from the outside environment by received messages it is not—as under the procedure invocation paradigm—dictated by the incoming messages. It is the object's autonomous “decision” if and when it is willing to respond to an incoming message.

Thus, the behavioral map may be considered as the “*sensory system*” [10] of the object which perceives the relevant environmental changes as well as the internal state changes of the object and initiates corresponding actions, i.e., operations which are declared in the **operations** part of the type definition.

3.1 The Basic Communication Concepts

Before discussing the computational model of the behavioral map we describe the basic communication primitives—discussed here from the sender's point of view. The receiver's reaction upon received messages is subsumed by the behavioral map (cf. Section 3.2) since it is only one of the possible stimuli upon which an autonomous object reacts.

A message is a string, in the following referred to as $\langle \text{msg_name} \rangle$, associated with an arbitrary number of arguments. A message is only meaningful if the receiver has an appropriate guard to “handle” the received message. A message can be relayed to an object using two different message passing paradigms. We have *synchronous message passing* (**call**) and *asynchronous message passing* (**send**). The synchronous message passing makes only sense if the message requests the invocation of some operation within the receiver. In this case the sender (caller) of the message waits until the receiver finishes processing the requested operation. The result, if any, of the operation invocation is returned to the sender and may be assigned to some properly typed variable within the sender object (variable r below). The syntax for synchronous message passing is:

$$[r :=] \text{ call } \langle \text{receiver object} \rangle \langle \text{msg_name} \rangle (e_1, \dots, e_n)$$

The synchronous message passing involves the blocking of the caller; however it does not—like in procedure invocation—warrant that the callee immediately answers the **call**. This would definitely violate object autonomy since it involves an outside dictation of the callee's behavior. Rather, the callee may decide to delay the call for some reasons, e.g., because some other—more important—stimuli are pending.

Any operation that can be invoked synchronously may, alternatively, be requested by asynchronous message passing. In this case the sender does not have to wait for receiving back the control from the receiver. After initiating the **send** the sender can immediately resume its activity and “work on other issues”. If the requested operation has a return value other than **void** then two possibilities exist:

1. the return value is, by default, discarded by the $\langle \text{receiver object} \rangle$, that is, the sender will never be aware of the outcome of the operation. It will not even know whether the operation was successfully executed at all.
2. the sender may optionally specify some message name via which the return value can be relayed back to it by the receiver. Again, this message name $\langle \text{msg_name}' \rangle$ can be any string. But of course, the sender object must contain some matching guard in order to accept the result.

The syntactical construct for asynchronous message passing is as follows:

$$\text{send } \langle \text{receiver object} \rangle \langle \text{msg_name} \rangle (e_1, \dots, e_n) \\ [\text{result via } \langle \text{msg_name}' \rangle]$$

The **result via** clause is—as described above—optional. Only if it is stated is the result of the requested operation returned via (an implicit) **send** of a message called $\langle \text{msg_name}' \rangle$ initiated by the $\langle \text{receiver object} \rangle$ after executing the requested operation. Otherwise, the result is discarded and only the side effects of the executed operation will materialize in the receiver.

3.2 The Behavioral Map

For autonomous objects the type definition of an object is enhanced by a **behavioral map** part. Consequently the complete type definition template then looks as shown in Figure 1.

The **behavioral map** constitutes a collection of guarded commands, similar to those proposed by Dijkstra [6]. More specifically we distinguish three types of guards:

1. $\langle \text{on guard} \rangle$ s, which handle requests in the form of received messages

```

[persistent] type ⟨type name⟩
  supertype ⟨type name⟩ is
  body
  ...
  operations
  ...
  behavioral map
    ⟨guard⟩1 [priority ⟨prio⟩1] do ⟨action⟩1
    ...
    ⟨guard⟩i [priority ⟨prio⟩i] do ⟨action⟩i;
    ...
    ⟨guard⟩n [priority ⟨prio⟩n] do ⟨action⟩n
  implementation
  ...
end type ⟨type name⟩;

```

Figure 1: Template of Autonomous Object Types

2. ⟨at guard⟩s, which handle time triggered events
3. ⟨if guard⟩s, which initiate activities dependent on the detection of certain distinguished states.

As mentioned before, we currently support one fixed computational model for autonomous objects. The subsequent discussion of the behavioral map constitutes only its semantics—is it this not necessarily a description of its actual implementation.

As outlined in the Introduction autonomous objects can be thought of as sequential processes in the sense that a once started operation cannot be interrupted by incoming requests and/or state changes. During their whole lifetime autonomous objects are active what can be expressed by the following infinite loop:

```

while alive do
  begin
    select ⟨guard⟩i;
    if ⟨guard⟩i is passable then ⟨action⟩i;
  end;
end;

```

Of course, this cannot be a realization because the availability of unlimited resources would be necessary. An actual implementation has to be optimized in such a way that objects can be deactivated in those time periods where no relevant events occur. But an efficient realization is beyond the scope of this paper.

As can be seen in the pseudo code section above an autonomous object constantly selects a guard, tests if it is passable and, depending on the result of this test, executes the respective action. We will first describe how a guard is selected for evaluation and then give a detailed description of the different possible behavioral map entries.

3.2.1 Indeterministic Selection

A guard is chosen for evaluation indeterministically. The user may influence the indeterministic selection of the guards by assigning integer priorities which are used as relative measures to prioritize certain guards over others. The guards are thus—indeterministically—selected for evaluation as indicated by the **priority** clauses. If no priority is given then the priority is assumed to be 1. Note, that this implies that the same impassable guard may be chosen twice or more times in sequence.

More precisely we have the following meaning. Given n entries in the behavioral map—as shown in Figure 1—where the i -th entry is assigned the priority ⟨prio⟩ _{i} we calculate the sum $P := \sum_{i=1}^n \langle \text{prio} \rangle_i$. We further define $P_k := \sum_{i=1}^k \langle \text{prio} \rangle_i$ for $1 \leq k \leq n$. The evaluation of the behavioral map can then be refined as follows:

```

while alive do
  begin
     $c := \text{random}(1, P);$   !! random between 1 and  $P$ .
    for  $i$  with  $P_{i-1} < c \leq P_i$  do
      if ⟨guard⟩ $i$  is passable then ⟨action⟩ $i$ ;
    end;
  end;
end;

```

The execution model can be thought of as playing roulette. The master process of an autonomous object constantly throws a ball in a roulette wheel whose number of slots equals the number of guards in the behavioral map. Each guard—irrespective of type (**on**, **at**, **if**-guard)—has an associated slot whose size is determined by its priority relative to the total priorities assigned. If the ball falls into the i th slot the ⟨guard⟩ _{i} is evaluated. If it is passable (see Section 3.2.2) the associated action is performed; otherwise the ball is thrown again.

Let us demonstrate the selection mechanism on the above introduced logistics control example. The **behavioral map** of the *BoltBox* could look as shown in Figure 2.

In this behavioral map the first guard is chosen 9 times more frequently than the second one, and the third one is chosen 90 (10) times more frequently than the second (first) one. This way, we give very high priority to incoming supplies in order to avoid emptying the storage space by processing requests while incoming supplies are waiting in the message list. Furthermore, incoming requests are carried out under different priorities: small consume requests, i.e., quantities not exceeding 20, are evaluated with priority 9, while all other consume requests—including the small ones—are evaluated with priority 1.

Now, we will have a closer look at the semantics of the different guards.

```

persistent type BoltBox is
  body
  ...
  operations
  ...
  behavioral map
    on consume(q) if q ≤ 20 and q ≤ self.Box.cardinality
      priority 9 do self.consume(q);
    on consume(q) if q ≤ self.Box.cardinality
      priority 1 do self.consume(q);
    on supply(items) priority 90 do self.supply(items);
  implementation
  ...
end type BoltBox;

```

Figure 2: Behavioral Map of the BoltBox

3.2.2 The Guards

In the beginning of this subsection we introduced three kinds of guards. Let us first discuss the **on** guard)s which initiate actions as a response to received messages from the outside environment, e.g., requests by other (client) objects or requests by the database user. The full syntax of the **on** guard) clause is defined as follows:

$$\langle \text{on guard} \rangle ::= \text{on } \langle \text{msg-name} \rangle (p_1, \dots, p_n) \\ \quad [\text{from } \langle \text{sender} \rangle] [\text{if } \langle \text{cond} \rangle]$$

All the **on** guard)s within the behavioral map share a single message buffer as indicated in Figure 3, into which all incoming messages are inserted—in the order in which they are received. If the chosen guard happens to be an **on** guard) this initiates a search in the message list for the *first* message from the beginning of the list (the bottom in Figure 3) for which the **on** clause including the **from** and the **if** part are satisfied. If one is found then the **do** part is executed and the message is deleted from the message list. If no such message is found in the entire message list the system selects another guard for evaluation. It should be stressed that the message list does not necessarily obey the FIFO (first in first out) order. Requests can be removed from any position in the message list.

For illustration reconsider the behavioral map of the *BoltBox* as specified in Figure 2. Note that this behavioral map prevents starvation of large *consume* requests because the **on** guards always search for the first qualifying entry within the message list starting at the beginning—and rearranging the message list is not allowed. Eventually every *consume* request—even a large one—moves to the beginning of the message list where it is then carried out by the second guard.

An **on** guard) evaluates to true if there is a pending message named $\langle \text{msg-name} \rangle$ with associated n pa-

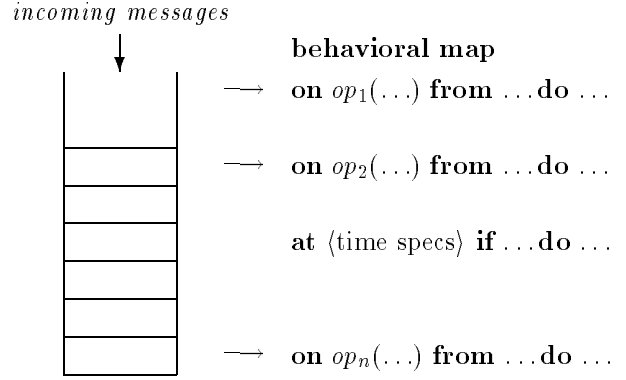


Figure 3: Graphical Representation of the **map**

rameters $p_1, \dots, p_n$ in the common message list. Note that this need not be the message at the bottom of the list. Additionally, the following conditions must hold:

1. if the **from** clause is specified the message $\langle \text{msg-name} \rangle$ has to be received from the specified $\langle \text{sender} \rangle$.
2. the—optional—**if** condition $\langle \text{cond} \rangle$ evaluates to true. The predicate $\langle \text{cond} \rangle$ may refer itself to the parameters of the message $\langle \text{msg-name} \rangle$.

If all conditions are satisfied the **on** guard) is said to be *passable*.

Aside from outside requests which are handled by the **on** clauses an autonomous object may also need to respond to time triggered actions. For this purpose the behavioral map may contain one or more **at** guard)s which have the following syntactical form:

$$\langle \text{at guard} \rangle ::= \text{at } \langle \text{date} \rangle [\text{every } \langle \text{duration} \rangle \\ \quad [\text{until } \langle \text{date} \rangle]] [\text{if } \langle \text{cond} \rangle]$$

The **at** guard) is evaluated analogously to the **on** guard), except that, now, the search for a matching message is replaced by checking whether the specified time for the next time triggered event has been reached. Of course, it cannot always be guaranteed that an **at** guard) is evaluated at exactly the specified time; therefore, the **at** clause evaluates to true if the specified time has been reached and the guard has not been passed in the meantime.

The **if** guard) constitutes the third—and last—kind of guards that our autonomous object model currently supports. These guards have the very simple syntactical form

$$\langle \text{if guard} \rangle ::= \text{if } \langle \text{cond} \rangle$$

This guard is *passable* if the condition $\langle \text{cond} \rangle$ evaluates to true. This allows the triggering of actions dependent on states of the object. If the **if** entry is also missing this can be read as **if true do** $\langle \text{action} \rangle$.

4 Advanced Communication Patterns

In Section 3.1 we introduced synchronous and asynchronous message passing as the basic message passing paradigms by which autonomous objects can communicate. Projecting these back onto processes this implies the existence of 1 : 1 channels between any two objects. In this section we argue that this is not sufficient for complex applications (see, e.g., [9]) because it sometimes obscures the control flow in complex applications with cooperating autonomous objects rather difficult. In many cases it leads to a more modular design if we could establish more dynamic $n : m$ channels via which cooperating objects communicate. In the first subsection we realize a particular $n : m$ channel, called the *RELAIS* that allows multicast (selective broadcast) of messages (in this case signals) to a dynamically determined set of objects. We show that the realization remains completely within our autonomous object framework as described so far.

Further, realizing communication patterns through autonomous objects allows to implement channels with arbitrary features. An example is given in the second subsection where we introduce an $n : m$ channel with an additional state which we call *FlipFlop*.

4.1 RELAIS Objects

In general, in a complex application many different objects may have to be “informed” of the occurrence of a particular event. For this purpose we introduce the autonomous object type *RELAIS* which is used to relate such relevant state changes among different objects. In this sense *RELAIS* objects serve as amplifiers which multiplex the flow of control within a complex application. This has two major advantages compared to direct sending of messages:

1. the system can be extended much more easily because the flow of control is no longer “hidden” in the (internal) object definition
2. complex applications can be built in a more modular fashion, such that logically related events can be handled by common *RELAIS* object.

A *RELAIS* object is raised explicitly by some object. Thereupon the *RELAIS* signals to all those objects which have notified the corresponding *RELAIS* that they are “interested” in perceiving the raising of the particular *RELAIS* object. This notification can take place at any time and thus supports easy extensibility. Graphically, the above outlined semantics of *RELAIS*s is depicted in Figure 4.

One of the left-hand side objects raises the *RELAIS* instance whereupon the right-hand side objects are

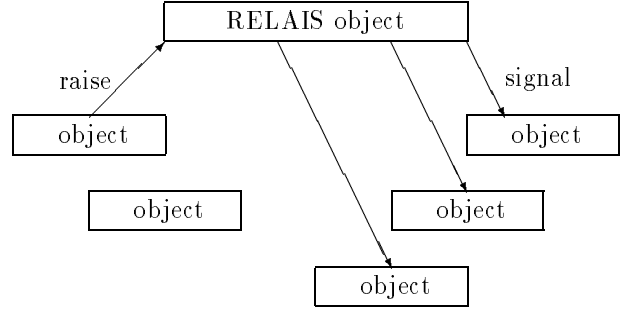


Figure 4: Flow of Control Through a *RELAIS* Object

sent the *signal* message informing them about the *raise*.

Note that the *RELAIS* type remains completely within our autonomous

object model—an indication for the expressiveness and extensibility of the language.

The *RELAIS* type is defined as follows.

persistent type *RELAIS* is

body

[ObjectsToNotify : {ANY} := {}]

operations

declare notify: ANY $\rightarrow$ void;

declare raise: $\rightarrow$ void code raiseNoArgs;

declare unnotify: ANY $\rightarrow$ void;

behavioral map

on raise do self.raise

on notify(oid) do self.notify(oid);

on unnotify(oid) do self.unnotify(oid);

implementation

define notify(oid) self.ObjectsToNotify.insert(oid);

define unnotify(oid) self.ObjectsToNotify.delete(oid);

define raiseNoArgs

foreach oid in self.ObjectsToNotify send oid signal;

end type *RELAIS*;

The semantics of the (autonomous) type *RELAIS* is as follows:

- upon receiving a *raise* message the *RELAIS* object “informs” all interested objects which are registered in the set *ObjectsToNotify* by sending them a *signal* message.
- an object *o* is included in the set *ObjectsToNotify* of a *RELAIS* object *e* as soon as the operation *notify(o)* has been executed within *e*, that is, the object *o* has to explicitly request the future notification. This clause, so to say, informs the *RELAIS* instance *e* that the object *o* is—from now on—interested in being informed whether the *RELAIS* *e* has been raised.
- the *unnotify* operation is used to cancel a formerly interested object from the *ObjectsToNotify* set.

The above introduced *raise* operator has no arguments; it is merely used to signal the occurrence

of an *event* via the particular *RELAIS*. However, sometimes it is useful to associate with such a signal some argument which describes the particular *event* in more detail. Such a one-argument *raise* operator is defined below:

```

persistent type RELAIS is
...
operations
  declare raise: ANY → void code raiseOneArg

implementation
  define raiseOneArg(Arg)
    foreach oid in self.ObjectsToNotify
      send oid signal(Arg)
end type RELAIS;

```

The *signal* message has been augmented such that it now relays the argument *Arg* that was passed in the *raise*. In general, one could—by overloading the *raise* operator—declare and define a *raise* operator that has an arbitrary number of of arbitrarily typed arguments.

4.2 The Autonomous Type FlipFlop

The autonomous type *RELAIS* can be utilized to *raise* and *signal* an event. However, it does not facilitate to maintain the *raised* event for a prolonged duration. Therefore, we introduce a subclass of *RELAIS*, called *FlipFlop*, which can “remember” whether it has been *raised*.

```

persistent type FlipFlop supertype RELAIS is
  body
    [state: BOOL := false; !! ObjectsToNotify is inh.
     ObjectsToNotifyOnReset: {ANY} := {}]
  operations
    declare reset: → void code resetFFNoArg;
    refine raise: → void code raiseFFNoArg;
    declare is_on: → BOOL;
    declare notifyOnReset: ANY → void;
    declare unnotifyOnReset: → void;
  behavioral map
    on raise do self.raise;
    on reset do self.reset;
    on notify(oid) do ...;
    ...
  implementation
    define is_on return self.state;
    define notifyOnReset(oid)
      self.ObjectsToNotifyOnReset.insert(oid);
    define unnotifyOnReset(oid)
      self.ObjectsToNotifyOnReset.delete(oid);
    define raiseFFNoArg
      begin
        self.state := true;
        foreach oid in self.ObjectsToNotify
          send oid signal;
        end define raiseFFNoArg;
    define resetFFNoArg

```

```

begin
  self.state := false;
  foreach oid in self.ObjectsToNotifyOnReset
    send oid unsignal;
  end define resetFFNoArg;
end type FlipFlop;

```

The *raise* operator which was inherited from the supertype *RELAIS* is *refined* here. The refined *raise* for the *FlipFlop* type not only informs “interested parties” but also sets the *state* variable to *true*. Like other object-oriented languages GOM dynamically binds refined operators in order to execute the most specific version according to the—dynamically determined—type of the receiver argument.

Additionally—to the *RELAIS* type—we defined the reset operator which signals all the interested objects the *unsignal* message upon receiving the *reset* message from some client object.

5 The Logistics Control Example Revisited

In this section we demonstrate how our concepts of autonomous objects and communication between them can be used to model our logistics control system in a manner that conforms directly to the dynamics of the application and thus is a very natural representation.

We create the following two *RELAIS* object—one is actually a *FlipFlop*—, i.e., global *RELAIS* variables, *OutOfBolts* and *BoltsReordered*:

```

var OutOfBolts: RELAIS;
    BoltsReordered: FlipFlop;
...
OutOfBolts.create;
BoltsReordered.create;

```

These two *RELAIS*s have the following semantics:

- *OutOfBolts* is raised whenever the quantity of still available bolts decreases below some minimum value, e.g., the quantity that is needed within a couple of days
- *BoltsReordered* is a *FlipFlop* that is set as soon as a reorder for new bolts has been issued.

We now indicate the design of the *BoltBox*, i.e., the shared autonomous GOM object that controls the bolts that are currently in stock.

```

persistent type BoltBox is
  body
    [MinQOH: int,
     Box: BoltSet]
  operations
    declare consume: int → BoltSet code consumeCode;
    declare supply: BoltSet → void code supplyCode;

```

```

behavioral map
on consume(quantity) if quantity ≤ self.Box.cardinality
  priority 1 do
    self.consume(quantity);
if self.Box.cardinality < self.MinQOH priority 1 do
  send OutOfBolts raise(QOH);
on supply(Bolts) priority 10 do
  begin
    self.supply(Bolts);
    send BoltsReordered reset;  ## notify the supply
  end;
implementation
define consumeCode(q)
  return self.Box.retrieveElems(q);
define supplyCode(Bolts) self.Box.union(Bolts);
end type BoltBox.

```

We have defined the autonomous *BoltBox* type such that it raises the event *OutOfBolts* whenever the still available quantity (*Box.cardinality*) is running low. We can now define the associated *Logistics-Control* object which—autonomously—takes care of reordering bolts whenever the stock is running low and of paying those new bolts when it is notified about their supplement.

```

persistent type LogisticsControl is
  body
    [BoltsOrder: int,
    ...]
  operations
    ...
  map
    on signal(quantity) from OutOfBolts
      if not (call BoltsReordered is-on) do
        begin
          send CentralPurchasing
            reorder("Bolts", BoltsOrder - quantity);
          send BoltsReordered raise;
        end;
    on unsignal from BoltsReordered do
      ...;  !! take care of payment
  implementation
    ...
end type LogisticsControl.

```

It would now be very easy to incorporate another agent object that is interested in detecting any shortages of *Bolts*. Assume, for example, that there is some chief (human) operator in the CIM environment who is in charge of controlling the logistics within the system—in order to interfere if anything goes wrong. For this purpose we may define a type *OperatorConsole* which displays any such abnormal state descriptions:

```

persistent type OperatorConsole is
  ...
  behavioral map
    on signal from OutOfBolts do
      display("Running out of bolts!");

```

```

...
end type OperatorConsole;

```

The *RELAIS* objects are used to multicast messages signalling the occurrence of a particular event in order to multiplex the flow of control. It is easy to incorporate yet another “channel” along which the information by requesting the *notify* operation within the *RELAIS* object. This makes it easy to extend existing applications without having to alter the internal implementation of autonomous objects. In our example, the operator console could be integrated into the system without augmenting the *BoltBox* definition.

6 Conclusion

The main hypothesis of this paper has been that autonomous objects which have variously been mentioned in the literature are an ideal means to model the highly complex and dynamic applications of technical environments. It turned out that, before doing so, a more precise meaning has to be given to the notion of autonomous object than has been the case so far. Basically, an autonomous object has been considered an object in the widely accepted sense that is associated with a permanent process which controls its own state transitions. It incorporates a set of guards that accept stimuli from outside or inside the object according to predefined conditions. Its co-operation follows a predefined computational model. Further, several communication mechanisms were introduced which support the interaction between a set of autonomous objects.

Precision in the definition of the basic notions, and provision of a simple but powerful conceptual framework for systems of objects are an essential prerequisite for developing a concise language through which to express the informational model of a given technical environment. Such a language containing autonomous objects offers an abstraction mechanism to model subsystems, e.g., subunits of an organization, in an independent manner. The integration of the subsystems can be done without superimposed control structures that tend to be very complex (sometimes too complex) and thus very difficult to realize in applications considered in this paper. A model implemented with that kind of language can be interpreted as a program for cooperating processes. Considerable part of the paper has been devoted to such a language and to a demonstration of its utility. Chief among the language constructs has been the concept of a behavioral map.

So far we have carried out only first, non-conclusive studies concerning *synchronization* of activities in our model. Based on the works by Liskov and Weihl [22, 21], Herlihy [8], and Schwartz and Spector [19]

it is relatively straightforward to incorporate transactions into our model by making the autonomous object types *atomic*, that is, equip the types participating in transactions with—their own localized—concurrency control schemes. However, the conventional transaction mechanism turns out too inflexible for long-running CIM applications; therefore, we have to work on more flexible concepts, like *Sagas* [7] and *ConTracts* [18], which should be provided as built-in concepts that can be customized by the user to suit his or her needs.

The autonomous objects we deal with so far are still a bit restricted: the computational model is the same for all objects and there is no intra-object parallelism. Also, the behavioral map is still too static. It should provide more flexibility, e.g., dynamically altering the priorities of the guards, in order to “fine-tune” the behavior of individual autonomous objects within a complex system.

Acknowledgements

We would like to acknowledge helpful discussions with G. Lausen, A. Reuter, A. Schill, N. Serbedzija, and H. Wächter. Our student E. Appel helped in implementing the experimental prototype.

References

- [1] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Ma, 1986.
- [2] G. R. Andrews. Synchronizing resources. *ACM Trans. Programming Languages and Systems*, 3(4):405–430, Oct 1981.
- [3] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 40–57, Kyoto, Japan, Dec 1989.
- [4] E. Casais. An object oriented system implementing KNOs. In *Proc. Conf. on Office Information Systems (COIS)*, pages 284–290, Palo Alto, Mar 1988.
- [5] J.F. Costa, A. Sernadas, and C. Sernadas. OBL-89 user's manual. Technical report, Instituto Superior Tecnico, Lisbon, 1989.
- [6] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug 75.
- [7] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, May 87.
- [8] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Systems*, 15(1):96–124, Mar 1990.
- [9] S. Jablonski. Kommunikationskonzepte für verteilte Datenverarbeitung. *Informatik Forschung und Entwicklung*, 4(3):149–159, 1989.
- [10] A. Kemper, P. C. Lockemann, G. Moerkotte, H.-D. Walter, and S. M. Lang. Autonomy over ubiquity: Coping with the complexity of a distributed world. In *Proc. Ninth Intl. Conf. on Entity Relationship Approach*, pages 281 – 298, Lausanne, Oct 1990.
- [11] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, Atlantic City, NJ, May 1990.
- [12] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, 1990.
- [13] A. Kemper and G. Moerkotte. Correcting anomalies of standard inheritance—a constraint based approach. In *Proc. of the Intl. Conf. on Database and Expert Systems Applications (DEXA)*, pages 49–55, Vienna, Austria, Aug 90. Springer-Verlag.
- [14] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Dublin, Ireland, Aug 1993.
- [15] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. der GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik-Fachberichte Nr. 270.
- [16] L. Liu and R. Mersman. Activity model: A declarative approach for capturing communication behaviour in object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 481–493, Vancouver, Canada, 1992.
- [17] O. M. Nierstrasz. Active objects in HYBRID. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, Oct 1987.
- [18] A. Reuter. Contracts: A means for extending control beyond transaction boundaries. Presentation at Third Workshop on High Performance Transaction Systems, Sep 89. Pacific Grove, CA.
- [19] P. M. Schwartz and A. Z. Spector. Synchronizing abstract types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.
- [20] D. Tsichritzis, E. Fiume, S. Gibbs, and O. Nierstrasz. KNOs: Knowledge acquisition, dissemination and manipulation objects. *ACM Trans. Office Information Systems*, 5(1):96–112, Jan 1987.
- [21] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Programming Languages and Systems*, 11(2):249–282, Apr 1989.
- [22] W. E. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Trans. Programming Languages and Systems*, 7(2):244–269, 1985.